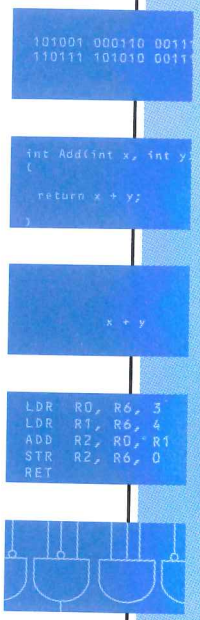
chapter

17



Recursion

17.1 Introduction

We start this chapter by describing a recursive procedure that you might already be familiar with. Suppose we want to find a particular student's exam in a set of exams that are already in alphabetical order. We might randomly examine the name on an exam about halfway through the set. If that randomly chosen exam is not the one we are looking for, we search the appropriate half using the very same technique. That is, we repeat the search on the first half or the second half, depending on whether the name we are looking for is less than or greater than the name on the exam at the halfway point. For example, say we are looking for Babe Ruth's exam and, at the halfway point, we find Mickey Mantle's exam. We then repeat the search on the second half of the original stack. Fairly quickly, we will locate Babe Ruth's exam, if it exists in the set. This technique of searching through a set of elements already in sorted order is *recursive*. We are applying the same searching algorithm to continually smaller and smaller subsets of exams.

The idea behind recursion is simple: A recursive function solves a task by calling itself on a smaller subtask. As we shall see, recursion is another way of expressing iterative program constructs. The power of recursion lies in its ability to elegantly capture the flow of control for certain tasks. There are some programming problems for which the recursive solution is far simpler than the corresponding solution using conventional iteration. In this chapter, we introduce you to the concept of recursion via five different examples. We examine how recursive functions are implemented on the LC-3. The elegance of the run-time stack mechanism is that recursive functions require no special handling—they

execute in the same manner as any other function. The main purpose of this chapter is to provide you with an initial but deep exposure to recursion so that you can analyze and reason about recursive programs. Being able to understand recursive code is a necessary ingredient for writing recursive code, and ultimately for recursion to become part of your problem-solving toolkit for attacking programming problems.

17.2 What Is Recursion?

A function that calls itself is a recursive function, as in the function `RunningSum` in Figure 17.1.

This function calculates the sum of all the integers between the input parameter `n` and 1. For example, `RunningSum(4)` calculates $4+3+2+1$. However, it does the calculation recursively. Notice that the running sum of 4 is really 4 plus the running sum of 3. Likewise, the running sum of 3 is 3 plus the running sum of 2. This *recursive* definition is the basis for a recursive algorithm. In other words,

$$\text{RunningSum}(n) = n + \text{RunningSum}(n-1)$$

In mathematics, we use *recurrence equations* to express such functions. The preceding equation is a recurrence equation for `RunningSum`. In order to complete the evaluation of this equation, we must also supply an initial case. So in addition to the preceding formula, we need to state

$$\text{RunningSum}(1) = 1$$

before we can completely evaluate the recurrence, which we do as follows:

$$\begin{aligned}\text{RunningSum}(4) &= 4 + \text{RunningSum}(3)\\ &= 4+3+\text{RunningSum}(2)\\ &= 4+3+2+\text{RunningSum}(1)\\ &= 4+3+2+1\end{aligned}$$

The C version of `RunningSum` works in the same manner as the recurrence equation. During execution of the function call `RunningSum(4)`, `RunningSum` makes a function call to itself, with an argument of 3 (i.e., `RunningSum(3)`). However, before `RunningSum(3)` ends, it makes a call to `RunningSum(2)`. And before `RunningSum(2)` ends, it makes a call to `RunningSum(1)`. `RunningSum(1)`, however, makes no additional recursive calls and returns the value 1 to

```
1  int RunningSum(int n)
2  {
3    if (n == 1)
4       return 1;
5    else
6       return (n + RunningSum(n-1));
7  }
```

Figure 17.1 A recursive function

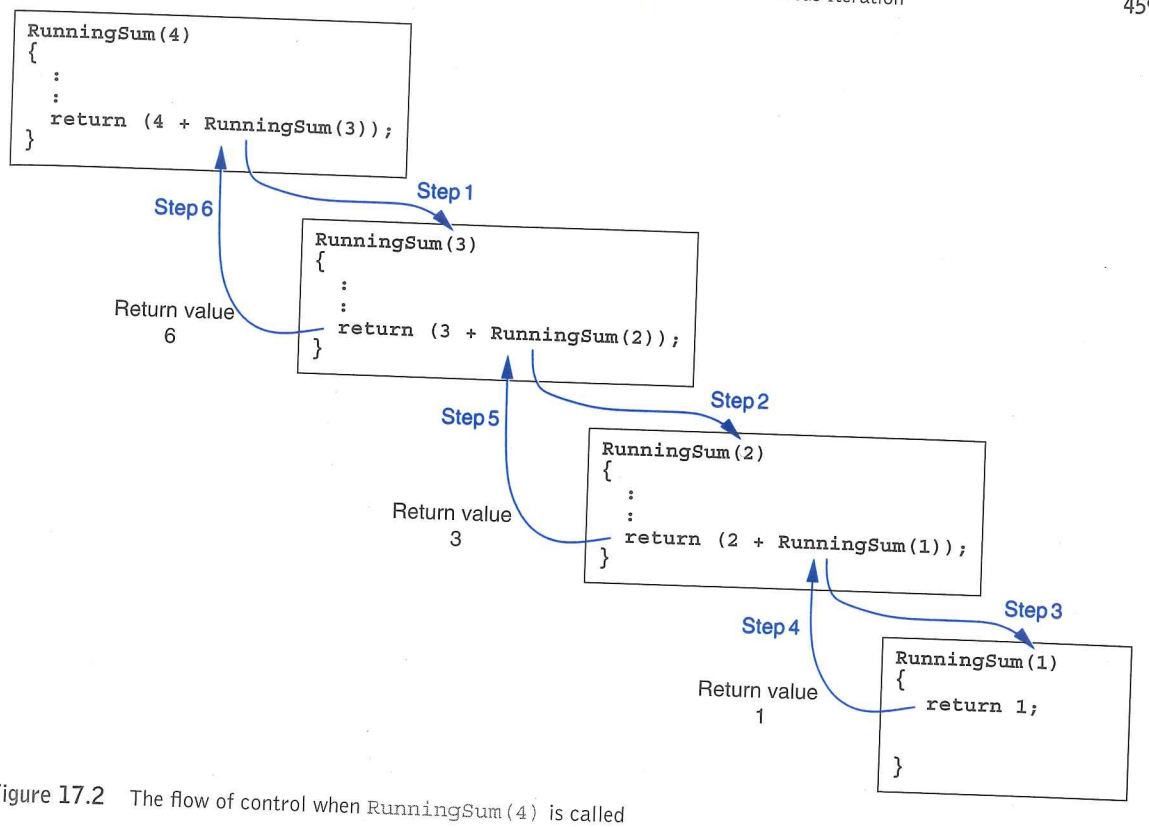


Figure 17.2 The flow of control when `RunningSum(4)` is called

`RunningSum(2)`, which enables `RunningSum(2)` to end, and return the value $2 + 1$ back to `RunningSum(3)`. This enables `RunningSum(3)` to end and pass a value of $3 + 2 + 1$ to `RunningSum(4)`. Figure 17.2 pictorially shows how the execution of `RunningSum(4)` proceeds.

17.3 Recursion versus Iteration

Clearly, we could have written `RunningSum` using a `for` loop, and the code would have been more straightforward than its recursive counterpart. We provided a recursive version here in order to demonstrate a recursive call in the context of an easy-to-understand example.

There is a parallel between using recursion and using conventional iteration (such as `for` and `while` loops) in programming. All recursive functions can be written using iteration. For certain programming problems, however, the recursive version is simpler and more elegant than the iterative version. Solutions to certain problems are naturally expressed in a recursive manner, such as problems that are expressed with recurrence equations. It is because of such problems that recursion is an indispensable programming technique. Knowing which problems require recursion and which are better solved with iteration is part of the art of computer programming; you will become better at when to use which with experience.

Recursion, as useful as it is, comes at a cost. As an experiment, write an iterative version of `RunningSum` and compare the running time for large n with the recursive version. To do this you can use library functions to get the time of day (for example, `gettimeofday`) before the function starts and when it ends. Plot the running time for a variety of values of n and you will notice that the recursive version is relatively slow (provided the compiler did not optimize away the recursion). As we shall see in Section 17.5, recursive functions incur function call overhead that iterative solutions do not.

17.4 Towers of Hanoi

One problem for which the recursive solution is the simpler solution is the classic puzzle Towers of Hanoi. The puzzle involves a platform with three posts. On one of the posts sit a number of wooden disks, each smaller than the one below it. The objective is to move all the disks from their current post to one of the other posts. However, there are two rules for moving disks: only one disk can be moved at a time, and a larger disk can never be placed upon a smaller disk. For example, Figure 17.3 shows a puzzle where five disks are on post 1. To solve this puzzle, these five disks must be moved to one of the other posts obeying the two rules.

As the legend associated with the puzzle goes, when the world was created, the priests at the Temple of Brahma were given the task of moving 64 disks from one post to another. When they completed their task, the world would end.

Now how would we go about writing a computer program to solve this puzzle? If we view the problem from the end first, we can make the following observation: the final sequence of moves **must** involve moving the largest disk from post 1 to the target post, say post 3, and then moving the other disks back on top of it. Conceptually, we need to move all $n - 1$ disks off the largest disk and onto the intermediate post, then move the largest disk from its post onto the target post. Finally, we move all $n - 1$ disks from the intermediate post onto the target post. And we are done! Actually, we are not quite done because moving $n - 1$ disks in one move is not legal. However, we have stated the problem in such a manner that we can solve it if we can solve the

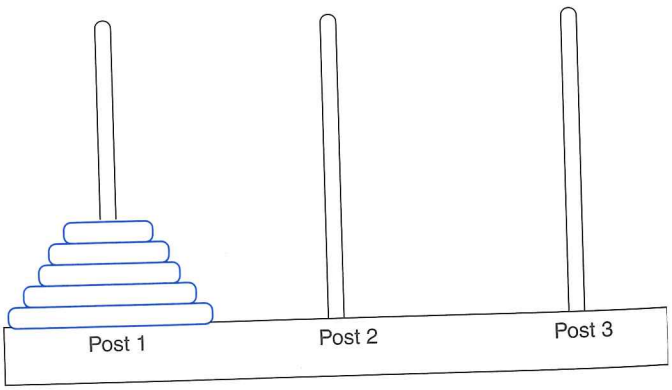


Figure 17.3 The Towers of Hanoi puzzle

```
/*
** Inputs
**   diskNumber is the disk to be moved (disk1 is smallest)
**   startPost is the post the disk is currently on
**   endPost is the post we want the disk to end on
**   midPost is the intermediate post
*/
MoveDisk(diskNumber, startPost, endPost, midPost)
{
   if (diskNumber > 1) {
       /* Move n-1 disks off the current disk on          */
       /* startPost and put them on the midPost           */
       MoveDisk(diskNumber-1, startPost, midPost, endPost);

       /* Move the largest disk.                          */
       printf("Move disk %d from post %d to post %d.\n",
              diskNumber, startPost, endPost);

       /* Move all n-1 disks from midPost onto endPost    */
       MoveDisk(diskNumber-1, midPost, endPost, startPost);
   }
   else
       printf("Move disk 1 from post %d to post %d.\n",
              startPost, endPost);
}
```

Figure 17.4 A recursive function to solve the Towers of Hanoi puzzle

two smaller subproblems of it. Once the largest disk is on the target post, we do not need to deal with it any further. Now the $n - 1^{th}$ disk becomes the largest disk, and the subobjective becomes to move it to the target pole. We can therefore apply the same technique but on a smaller subproblem.

We now have a recursive definition of the problem: In order to move n disks to the target post, which we symbolically represent as `Move(n, target)`, we first move $n - 1$ disks to the intermediate post—`Move(n-1, intermediate)`—then move the n^{th} disk to the target, and finally move $n - 1$ disks from the intermediate to the target, or `Move(n-1, target)`. So in order to `Move(n, target)`, two recursive calls are made to solve two smaller subproblems involving $n - 1$ disks.

As with recurrence equations in mathematics, all recursive definitions require a *base case*, which ends the recursion. In the way we have formulated the problem, the base case involves moving the smallest disk (disk 1). Moving disk 1 requires no other disks to be moved since it is always on top and can be moved directly from one post to any another without moving any other disks. Without a base case, a recursive function would have an infinite recursion, similar to an infinite loop in conventional iteration.

Taking our recursive definition to C code is fairly straightforward. Figure 17.4 is a recursive C function of this algorithm.

Let's see what happens when we play a game with three disks. Following is an initial function call to `MoveDisk`. We start off by saying that we want to move

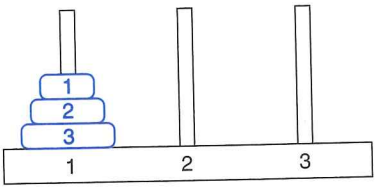


Figure 17.5 The Towers of Hanoi puzzle, initial configuration

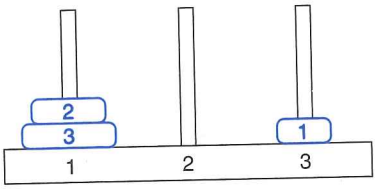


Figure 17.6 The Towers of Hanoi puzzle, after first move

disk 3 (the largest disk) from post 1 to post 3, using post 2 as the intermediate storage post. That is, we want to solve a three-disk Towers of Hanoi puzzle. See Figure 17.5.

```
/* diskNumber 3; startPost 1; endPost 3; midPost 2 */
MoveDisk(3, 1, 3, 2)
```

This call invokes another call to `MoveDisk` to move disks 1 and 2 off disk 3 and onto post 2 using post 3 as intermediate storage. The call is performed at line 15 in the source code.

```
/* diskNumber 2; startPost 1; endPost 2; midPost 3 */
MoveDisk(2, 1, 2, 3)
```

To move disk 2 from post 1 to post 2, we must first move disk 1 off disk 2 and onto post 3 (the intermediate post). So this triggers another call to `MoveDisk` again from the call on line 15.

```
/* diskNumber 1; startPost 1; endPost 3; midPost 2 */
MoveDisk(1, 1, 3, 2)
```

Since disk 1 can be directly moved, the second `printf` statement is executed. See Figure 17.6.

```
Move disk number 1 from post 1 to post 3.
```

Now, this invocation of `MoveDisk` returns to its caller, which was the call `MoveDisk(2, 1, 2, 3)`. Recall that we were waiting for all disks on top of disk 2 to be moved to post 3. Since that is now complete, we can now move disk 2 from post 1 to post 2. The `printf` is the next statement to execute, signaling another disk to be moved. See Figure 17.7.

```
Move disk number 2 from post 1 to post 2.
```

Next, a call is made to move all disks that were on disk 2 back onto disk 2. This happens at the call on line 22 of the source code for `MoveDisk`.

```
/* diskNumber 1; startPost 3; endPost 2; midPost 1 */
MoveDisk(1, 3, 2, 1)
```

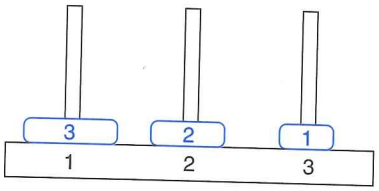


Figure 17.7 The Towers of Hanoi puzzle, after second move

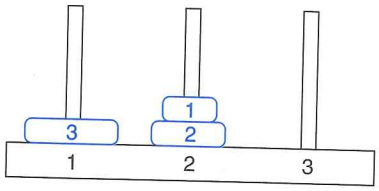


Figure 17.8 The Towers of Hanoi puzzle, after third move

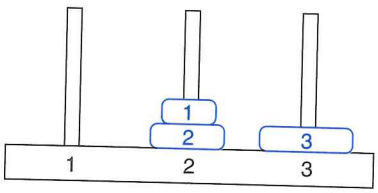


Figure 17.9 The Towers of Hanoi puzzle, after fourth move

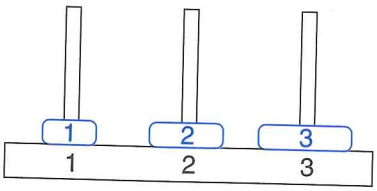


Figure 17.10 The Towers of Hanoi puzzle, after fifth move

Again, since disk 1 has no disks on top of it, we see the move printed. See Figure 17.8.

```
Move disk number 1 from post 3 to post 2.
```

Now control passes back to the call `MoveDisk(2, 1, 2, 3)` which, having completed its task of moving disk 2 (and all disks on top of it) from post 1 to post 2, returns to its caller. Its caller is `MoveDisk(3, 1, 3, 2)`. Now, all disks have been moved off disk 3 and onto post 2. Disk 3 can be moved from post 1 onto post 3. The `printf` is the next statement executed. See Figure 17.9.

```
Move disk number 3 from post 1 to post 3.
```

The next subtask remaining is to move disk 2 (and all disks on top of it) from post 2 onto post 3. We can use post 1 for intermediate storage. The following call occurs on line 22 of the source code.

```
/* diskNumber 2; startPost 2; endPost 3; midPost 1 */
MoveDisk(2, 2, 3, 1)
```

In order to do so, we must first move disk 1 from post 2 onto post 1. This call is made from line 15 in the source code.

```
/* diskNumber 1; startPost 2; endPost 1; midPost 3 */
MoveDisk(1, 2, 1, 3)
```

The move requires no submoves. See Figure 17.10.

```
Move disk number 1 from post 2 to post 1.
```

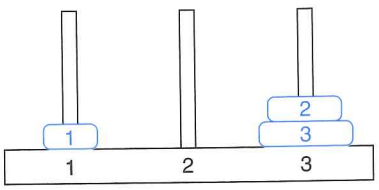


Figure 17.11 The Towers of Hanoi puzzle, after sixth move

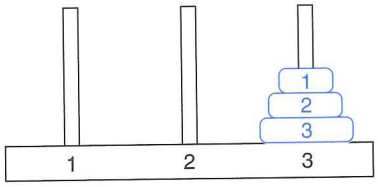


Figure 17.12 The Towers of Hanoi puzzle, completed

Return passes back to the caller `MoveDisk(2, 2, 3, 1)`, and disk 2 is moved onto post 3. See Figure 17.11.

```
Move disk number 2 from post 2 to post 3.
```

The only thing remaining is to move all disks that were on disk 2 back on top.

```
/* diskNumber 1; startPost 1; endPost 3; midPost 2 */
MoveDisk(1, 1, 3, 2)
```

The move is done immediately. See Figure 17.12.

```
Move disk number 1 from post 1 to post 3.
```

and the puzzle is completed!

Let's summarize the action of the recursion by examining the sequence of function calls that were made in solving the three-disk puzzle:

```
MoveDisk(3, 1, 3, 2)   /* Initial Call */
MoveDisk(2, 1, 2, 3)
MoveDisk(1, 1, 3, 2)
MoveDisk(1, 2, 3, 1)
MoveDisk(2, 2, 3, 1)
MoveDisk(1, 2, 1, 3)
MoveDisk(1, 1, 3, 2)
```

Consider how you would write an iterative version of a program to solve this puzzle and you will appreciate the simplicity of the recursive version. Returning to the legend of the Towers of Hanoi: the world will end when the monks finish solving a 64-disk version of the puzzle. If each move takes one second, how long will it take the monks to solve the puzzle?

17.5 Fibonacci Numbers

The following recurrence equations generate a well-known sequence of numbers called the *Fibonacci numbers*, which has some interesting mathematical, geometrical, and natural properties.

$$f(n) = f(n-1) + f(n-2)$$
$$f(1) = 1$$
$$f(0) = 1$$

In other words, the n^{th} Fibonacci number is the sum of the previous two. The series is 1, 1, 2, 3, 5, 8, 13, ... This series was first formulated by the Italian mathematician Leonardo of Pisa around the year 1200. His father's name was Bonacci, thus he often called himself Fibonacci as a shortening of *filius Bonacci*, or son of Bonacci. Fibonacci formulated this series as a way of estimating breeding rabbit populations, and we have since discovered some facinating ways in which the series models some other natural phenomena such as the structure of a spiral shell or the pattern of petals on a flower.

We can formulate a recursive function to calculate the n^{th} Fibonacci number directly from the recurrence equations. `Fibonacci(n)` is recursively calculated by `Fibonacci(n-1) + Fibonacci(n-2)`. The base case of the recursion is simply the fact that `Fibonacci(1)` and `Fibonacci(0)` both equal 1. Figure 17.13 lists the recursive code to calculate the n^{th} Fibonacci number.

```
1  #include <stdio.h>
2
3  int Fibonacci(int n);
4
5  int main()
6  {
7    int in;
8    int number;
9
10   printf("Which Fibonacci number? ");
11   scanf("%d", &in);
12
13   number = Fibonacci(in);
14   printf("That Fibonacci number is %d\n", number);
15 }
16
17 int Fibonacci(int n)
18 {
19   int sum;
20
21   if (n == 0 || n == 1)
22      return 1;
23   else {
24      sum = (Fibonacci(n-1) + Fibonacci(n-2));
25      return sum;
26   }
27 }
```

Figure 17.13 `Fibonacci` is a recursive C function to calculate the n^{th} Fibonacci number

We will use this example to examine how recursion works from the perspective of the lower levels of the computing system. In particular, we will examine the run-time stack mechanism and how it deals with recursive calls. Whenever the function is called, whether from itself or another function, a new copy of its activation record is pushed onto the run-time stack. That is, each invocation of the function gets a new, private copy of parameters and local variables, where each copy is different than any other copy. This must be the case in order for recursion to work, and the run-time stack enables this. If the variables of this function were statically allocated in memory, each recursive call to `Fibonacci` would overwrite the values of the previous call.

Let's see what happens when we call the function `Fibonacci` with the parameter 3, `Fibonacci(3)`. We start off with the activation record for `Fibonacci(3)` on top of the run-time stack. Figure 17.14 shows the progression of the stack as the original function call is evaluated.

The function call `Fibonacci(3)` will calculate first `Fibonacci(3-1)`, as the expression `Fibonacci(n-1) + Fibonacci(n-2)` is evaluated left to right. Therefore, a call is first made to `Fibonacci(2)`, and an activation record for `Fibonacci(2)` is pushed onto the run-time stack (see Figure 17.14, step 2).

For `Fibonacci(2)`, the parameter `n` equals 2 and does not meet the terminal condition, therefore a call is made to `Fibonacci(1)` (see Figure 17.14, step 3). This call is made in the course of evaluating `Fibonacci(2-1) + Fibonacci(2-2)`.

The call `Fibonacci(1)` results in no more recursive calls because the parameter `n` meets the terminal condition. The value 1 is returned to `Fibonacci(2)`, which now can complete the evaluation of `Fibonacci(1) + Fibonacci(0)` by calling `Fibonacci(0)` (see Figure 17.14, step 4). The call `Fibonacci(0)` immediately returns a 1.

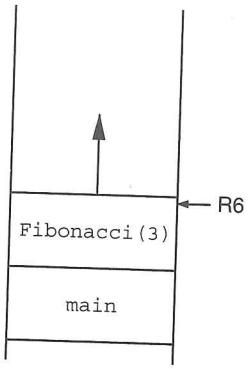
Now, the call `Fibonacci(2)` can complete and return its subcalculation (its result is 2) to its caller, `Fibonacci(3)`. Having completed the left-hand component of the expression `Fibonacci(2) + Fibonacci(1)`, `Fibonacci(3)` calls `Fibonacci(1)` (see Figure 17.14, step 5), which immediately returns the value 1. Now `Fibonacci(3)` is done—its result is 3 (Figure 17.14, step 6).

We could state the recursion of `Fibonacci(3)` algebraically, as follows:

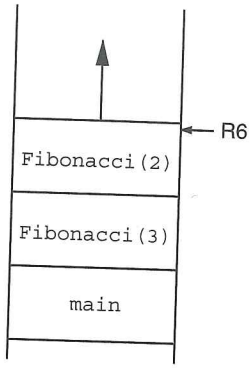
```
Fibonacci(3) = Fibonacci(2) + Fibonacci(1)
             = (Fibonacci(1) + Fibonacci(0)) + Fibonacci(1)
             = 1 + 1 + 1 = 3
```

The sequence of function calls made during the evaluations of `Fibonacci(3)` is as follows:

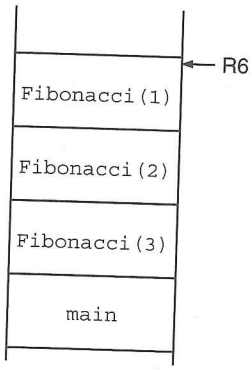
```
Fibonacci(3)
Fibonacci(2)
Fibonacci(1)
Fibonacci(0)
Fibonacci(1)
```



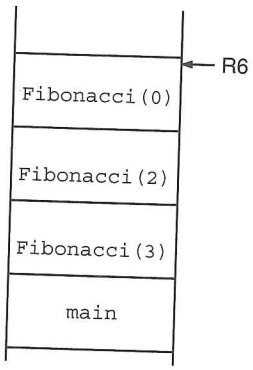
Step 1: Initial call



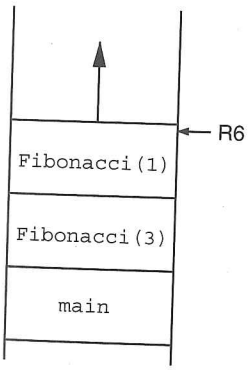
Step 2: `Fibonacci(3)` calls `Fibonacci(2)`



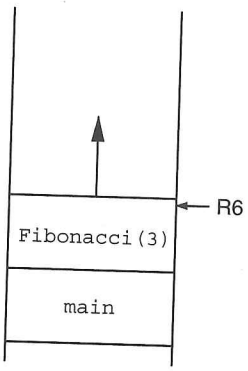
Step 3: `Fibonacci(2)` calls `Fibonacci(1)`



Step 4: `Fibonacci(2)` calls `Fibonacci(0)`



Step 5: `Fibonacci(3)` calls `Fibonacci(1)`



Step 6: Back to the starting point

Figure 17.14 Snapshots of the run-time stack for the function call `Fibonacci(3)`

Walk through the execution of `Fibonacci(4)` and you will notice that the sequence of calls made by `Fibonacci(3)` is a subset of the calls made by `Fibonacci(4)`. No surprise, since `Fibonacci(4) = Fibonacci(3) + Fibonacci(2)`. Likewise, the sequence of calls made by `Fibonacci(4)` is a subset of the calls made by `Fibonacci(5)`. There is an exercise at the end of this chapter involving calculating the number of function calls made during the evaluation of `Fibonacci(n)`.

The LC-3 C compiler generates the following code for this program, listed in Figure 17.15. Notice that no special treatment was required because this function is recursive. Because of the run-time stack mechanism for activating functions, a recursive function gets treated like every other function. If you examine this code closely, you will notice that the compiler generated a temporary variable in order to translate line 24 of `Fibonacci` properly. Most compilers will generate such temporaries when compiling complex expressions. Such temporary values are allocated storage in the activation on top of the space for the programmer-declared local variables.

17.6 Binary Search

In the introduction to this chapter, we described a recursive technique for finding a particular exam in a set of exams that are in alphabetical order. The technique is called *binary search*, and it is a very rapid way of finding a particular element within a list of elements in sorted order. At this point, given our understanding of recursion and of arrays, we can specify a recursive function in C to perform binary search.

Say we want to find a particular integer value in an array of integers that is in ascending order. The function should return the index of the integer, or a -1 if the integer does not exist. To accomplish this, we will use the binary search technique as such: given an array and an integer to search for, we will examine the midpoint of the array and determine if the integer is (1) equal to the value at the midpoint, (2) less than the value at the midpoint, or (3) greater than the value at the midpoint. If it is equal, we are done. If it is less than, we perform the search again, but this time only on the first half of the array. If it is greater than, we perform the search only on the second half of the array. Notice that we can express cases (2) and (3) using recursive calls. But what happens if the value we are searching for does not exist within the array? Given this recursive technique of performing searches on smaller and smaller subarrays of the original array, we eventually perform a search on an array that has no elements (e.g., of size 0) if the item we are searching for does not exist. If we encounter this situation, we will return a -1. This will be a base case in the recursion.

Figure 17.16 contains the recursive implementation of the binary search algorithm in C. Notice that in order to determine the size of the array at each step, we pass the starting point and ending point of the subarray along with each call to `BinarySearch`. Each call refines the variables `start` and `end` to search smaller and smaller subarrays of the original array `list`.

```
1  Fibonacci:
2    ADD  R6, R6, #-2 ; push return value/address
3    STR  R7, R6, #0  ; store return address
4    ADD  R6, R6, #-1 ; push caller's frame pointer
5    STR  R5, R6, #0  ;
6    ADD  R5, R6, #-1 ; set new frame pointer
7    ADD  R6, R6, #-2 ; allocate space for locals and temps
8
9    LDR  R0, R5, #4  ; load the parameter n
10   BRZ  FIB_BASE    ; n==0
11   ADD  R0, R0, #-1 ;
12   BRZ  FIB_BASE    ; n==1
13
14   LDR  R0, R5, #4  ; load the parameter n
15   ADD  R0, R0, #-1 ; calculate n-1
16   ADD  R6, R6, #-1 ; push n-1
17   STR  R0, R6, #0  ;
18   JSR  Fibonacci   ; call to Fibonacci(n-1)
19
20   LDR  R0, R6, #0  ; read the return value at top of stack
21   ADD  R6, R6, #-1 ; pop return value
22   STR  R0, R5, #-1 ; store it into temporary value
23   LDR  R0, R5, #4  ; load the parameter n
24   ADD  R0, R0, #-2 ; calculate n-2
25   ADD  R6, R6, #-1 ; push n-2
26   STR  R0, R6, #0  ;
27   JSR  Fibonacci   ; call to Fibonacci(n-2)
28
29   LDR  R0, R6, #0  ; read the return value at top of stack
30   ADD  R6, R6, #-1 ; pop return value
31   LDR  R1, R5, #-1 ; read temporary value: Fibonacci(n-1)
32   ADD  R0, R0, R1  ; Fibonacci(n-1) + Fibonacci(n-2)
33   BR   FIB_END     ; branch to end of code
34
35 FIB_BASE:
36   AND  R0, R0, #0  ; clear R0
37   ADD  R0, R0, #1  ; R0 = 1
38
39 FIB_END:
40   STR  R0, R5, #3  ; write the return value
41   ADD  R6, R5, #1  ; pop local variables
42   LDR  R5, R6, #0  ; restore caller's frame pointer
43   ADD  R6, R6, #1  ;
44   LDR  R7, R6, #0  ; pop return address
45   ADD  R6, R6, #1  ;
46   RET
```

Figure 17.15 Fibonacci in LC-3 assembly code

```
1  /*
2  ** This function returns the position of 'item' if it exists
3  ** between list[start] and list[end], or -1 if it does not.
4  */
5  int BinarySearch(int item, int list[], int start, int end)
6  {
7    int middle = (end + start) / 2;
8
9    /* Did we not find what we are looking for? */
10   if (end < start)
11     return -1;
12
13   /* Did we find the item? */
14   else if (list[middle] == item)
15     return middle;
16
17   /* Should we search the first half of the array? */
18   else if (item < list[middle])
19     return BinarySearch(item, list, start, middle - 1);
20
21   /* Or should we search the second half of the array? */
22   else
23     return BinarySearch(item, list, middle + 1, end);
24 }
```

Figure 17.16 A recursive C function to perform binary search

Figure 17.17 provides a pictorial representation of this code during execution. The array `list` contains 11 elements as shown. The initial call to `BinarySearch` passes the value we are looking for (`item`) and the array to be searched (recall from Chapter 16 that this is the address of the very first element, or base address, of the array). Along with the array, we provide the *extent* of the array. That is, we provide the starting point and ending point of the portion of the array to be searched. In every subsequent recursive call to `BinarySearch`, this extent is made smaller, eventually reaching a point where the subset of the array we are searching has either only one element or no elements at all. These two situations are the base cases of the recursion.

Instead of resorting to a technique like binary search, we could have attempted a more straightforward sequential search through the array. That is, we could examine `list[0]`, then `list[1]`, then `list[2]`, etc., and eventually either find the item or determine that it does not exist. Binary search, however, will require fewer comparisons and can potentially execute faster if the array is large enough. In subsequent computing courses you will analyze binary search and derive that its running time is proportional to $\log_2 n$, where n is the size of the array. Sequential search, on the other hand, is proportional to n.

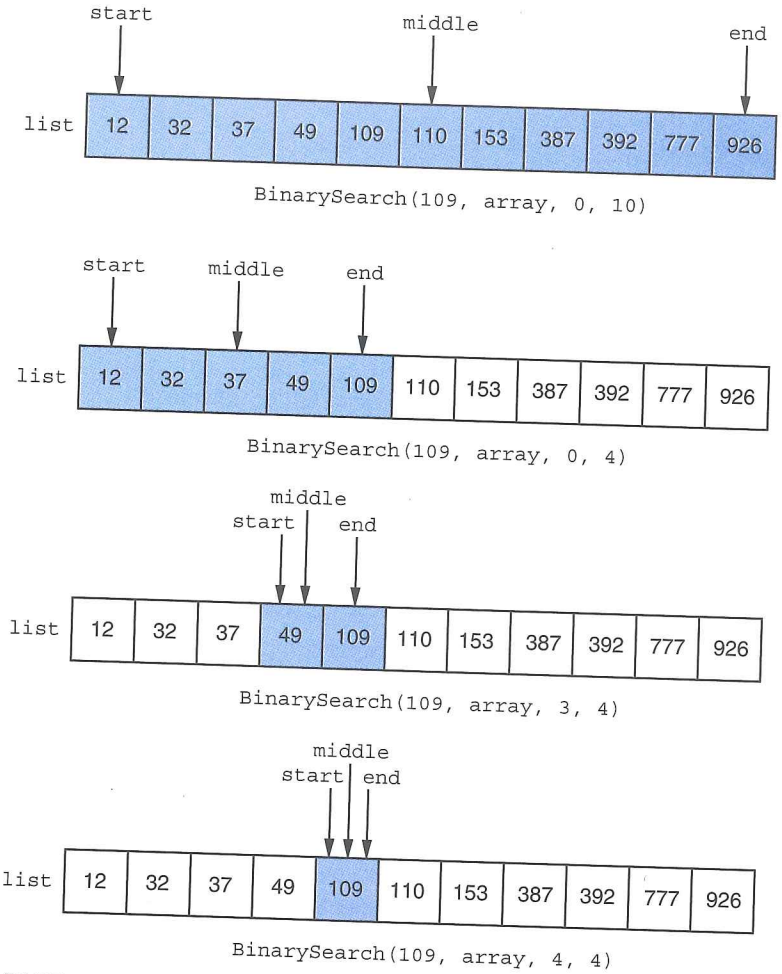


Figure 17.17 `BinarySearch` performed on an array of 11 elements. We are searching for the element 109

17.7 Integer to ASCII

Our final example of a recursive function is a function that converts an arbitrary integer value into a string of ASCII characters. Recall from Chapter 10 that in order to display an integer value on the screen, each digit of the value must be individually extracted, converted into ASCII, and then displayed on the output device. In Chapter 10, we wrote an LC-3 routine to do this using a straightforward iterative technique.

We can do this recursively with the following recursive formulation: if the number to be displayed is a single digit, we convert it to ASCII and display it and we are done (base case). If the number is multiple digits, we make a recursive

```
1  #include <stdio.h>
2
3  void IntToAscii(int i);
4
5  int main()
6  {
7    int in;
8
9    printf("Input number: ");
10   scanf("%d", &in);
11
12   IntToAscii(in);
13   printf("\n");
14 }
15
16 void IntToAscii(int num)
17 {
18   int prefix;
19   int currDigit;
20
21   if (num < 10)                    /* The terminal case      */
22     printf("%c", num + '0');
23   else {
24     prefix = num / 10;             /* Convert the number     */
25     IntToAscii(prefix);            /* without last digit     */
26
27     currDigit = num % 10;          /* Then print last digit */
28     printf("%c", currDigit + '0');
29   }
30 }
```

Figure 17.18 `IntToAscii` is a recursive function that converts a positive integer to ASCII

call on the number without the least significant (rightmost) digit, and when the recursive call returns we display the rightmost digit.

Figure 17.18 lists the recursive C function. It takes a positive integer value and converts each digit of the value into ASCII and displays the resulting characters.

The recursive function `IntToAscii` works as follows: to print out a number, say 21,669, for example (i.e., we are making the call `IntToAscii(21669)`), the function will subdivide the problem into two parts. First 2166 must be printed out via a recursive call to `IntToAscii`, and once the call is done, the 9 will be printed.

The function removes the least significant digit of the parameter `num` by shifting it to the right one digit by dividing by 10. With this new (and smaller) value, we make a recursive call. If the input value `num` is only a single digit, it is converted to ASCII and displayed to the screen—no recursive calls necessary for this case.

Once control returns to each call, the digit that was removed is converted to ASCII and displayed. To clarify, we present the series of calls for the original call of `IntToAscii(12345)`:

```
IntToAscii(12345)
IntToAscii(1234)
IntToAscii(123)
IntToAscii(12)
IntToAscii(1)
printf('1')
printf('2')
printf('3')
printf('4')
printf('5')
```

17.8 Summary

In this chapter, we introduced the concept of recursion. We can solve a problem recursively by using a function that calls itself on smaller subproblems. With recursion, we state the function, say $f(n)$, in terms of the same function on smaller values of n, say for example, $f(n-1)$. The Fibonacci series, for example, is recursively stated as

```
Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2);
```

For the recursion to eventually terminate, recursive calls require a base case.

Recursion is a powerful programming tool that, when applied to the right problem, can make the task of programming considerably easier. For example, the Towers of Hanoi puzzle can be solved in a simple manner with recursion. It is much harder to formulate using iteration. In future courses, you will examine ways of organizing data involving pointers (e.g., trees and graphs) where the simplest techniques to manipulate the data structure involve recursive functions. At the lower levels, recursive functions are handled in exactly the same manner as any other function call. The run-time stack mechanism enables this by allowing us to allocate in memory an activation record for each function invocation so that it does not conflict with any other invocation's activation record.

Exercises

17.1 For these questions, refer to the examples that appear in the chapter.

a. How many calls to `RunningSum` (see Section 17.2) are made for the call `RunningSum(10)`?

b. How about for the call `RunningSum(n)`? Give your answer in terms of n.

c. How many calls to `MoveDisk` are made in the Towers of Hanoi problem if the initial call is `MoveDisk(4, 1, 3, 2)`? This call plays out a four-disk game.

d. How many calls are made for an n-disk game?

e. How many calls to `Fibonacci` (see Figure 17.13) are made for the initial call `Fibonacci(10)`?

f. How many calls are required for the n^{th} Fibonacci number?

17.2 Is the return address for a recursive function always the same at each function call? Why or why not?

17.3 What would happen if we swapped the `printf` call with the recursive call in the code for `IntToAscii` in Figure 17.18?

17.4 What does the following function produce for `count(20)`?

```
int count(int arg)
{
   if (arg < 1)
      return 0;

   else if (arg % 2)
      return(1 + count(arg - 2));
   else
      return(1 + count(arg - 1));
}
```

17.5 Consider the following C program:

```
#include <stdio.h>

int Power(int a, int b);

int main(void)
{
   int x, y, z;

   printf("Input two numbers: ");
   scanf("%d %d", &x, &y);

   if (x > 0 && y > 0)
     z = Power(x,y);
   else
     z = 0;

   printf("The result is %d.\n", z);
}

int Power(int a, int b)
{
   if (a < b)
      return 0;
   else
      return 1 + Power(a/b, b);
}
```

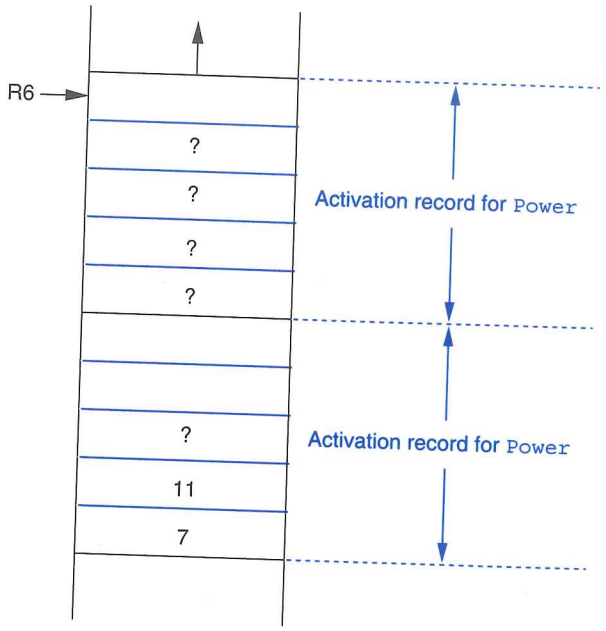


Figure 17.19 Run-time stack after function `Power` is called

a. State the complete output if the input is
(1) `4 9`
(2) `27 5`
(3) `-1 3`

b. What does the function `Power` compute?

c. Figure 17.19 is a snapshot of the stack after a call to the function `Power`. Two activation records are shown, with some of the entries filled in. Assume the snapshot was taken just before execution of one of the `return` statements in `Power`. What are the values in the entries marked with a question mark? If an entry contains an address, use an arrow to indicate the location the address refers to.

17.6 Consider the following C function:

```
int Sigma( int k )
{
   int l;

   l = k -1;

   if (k==0)
      return 0;
   else
      return (k + Sigma(l));
}
```

a. Convert the recursive function into a nonrecursive function. Assume `Sigma()` will always be called with a nonnegative argument.

b. Exactly 1 KB of contiguous memory is available for the run-time stack, and addresses and integers are 16 bits wide. How many recursive function calls can be made before the program runs out of memory? Assume no storage is needed for temporary values.

17.7 The following C program is compiled and executed on the LC-3. When the program is executed, the run-time stack starts at memory location xFEFF and grows toward xC000 (the stack can occupy up to 16 KBytes of memory).

```
SevenUp(int x)
{
   if (x == 1)
      return 7;
   else
      return (7 + sevenUp(x - 1));
}

int main()
{
   int a;

   printf("Input a number \n");
   scanf("%d", &a);

   a = SevenUp(a);

   printf("%d is 7 times the number\n", a);
}
```

a. What is the largest input value for which this program will run correctly? Explain your answer.

b. If the run-time stack can occupy only 4 KBytes of memory, what is the largest input value for which this program will run correctly? Explain your answer.

17.8 Write an iterative version of a function to find the n^{th} Fibonacci number. Plot the running time of this iterative version to the running time of the recursive version on a variety of values for n. Why is the recursive version significantly slower when n is sufficiently large?

17.9 The binary search routine shown in Figure 17.16 searches through an array that is in ascending order. Rewrite the code so that it works for arrays in descending order.

17.10 Following is a very famous algorithm whose recursive version is significantly easier to express than the iterative one. For the following subproblems, provide the final value returned by the function.

```
int ea(int x, int y)
{
   int a;

   if (y == 0)
      return x;
   else {
      a = x % y;
      return (ea(y, a));
   }
}
```

a. `ea(12, 15)`
b. `ea(6, 10)`
c. `ea(110, 24)`
d. What does this function calculate? Consider how you might construct an iterative version to calculate the same thing.

17.11 Write a program without recursive functions equivalent to the following C program.

```
int main()
{
   printf("%d", M());
}

void M()
{
   int num, x;
   printf("Type a number: ");
   scanf("%d", &num);
   if (num <= 0)
      return 0;
   else {
      x = M();
      if (num > x)
         return num;
      else
         return x;
   }
}
```

17.12 Consider the following recursive function:

```
int func (int arg)
{
   if (arg % 2 != 0)
      return func(arg - 1);
   if (arg <= 0)
      return 1;

   return func(arg/2) + 1;
}
```

a. Is there a value of `arg` that causes an infinite recursion? If so, what is it?

b. Suppose that the function `func` is part of a program whose `main` function follows. How many function calls are made to `func` when the program is executed?

```
int main()
{
   printf("The value is %d\n", func(10));
}
```

c. What value is output by the program?

17.13 The following function is a recursive function that takes a string of characters of unknown length and determines if it contains balanced parentheses. The function `Balanced` is designed to match parentheses. It returns a 0 if the parentheses in the character array string are balanced and a nonzero value if the parentheses are not balanced. The initial call to `Balanced` would be: `Balanced(string, 0, 0);`

The function `Balanced` that follows, however, is missing a few key pieces of code. Fill in the three underlined missing portions in the code.

```
int Balanced(char string[], int position, int count)
{
  if (_____)
    return count;

  else if (string[position] == _____)
    return Balanced( string, ++position, ++count);

  else if (string[position] == _____)
    return Balanced( string, ++position, --count);

  else
    return Balanced( string, ++position, count);
}
```

17.14 What is the output of the following C program?

```
#include <stdio.h>

void Magic(int in);
int Even(int n);

int main()
{
   Magic(10);
}

void Magic(int in)
{
   if (in == 0)
      return;
   if (Even(in))
      printf("%i\n", in);
   Magic(in - 1);
   if (!Even(in))
      printf("%i\n", in);
   return;
}

int Even(int n)
{
   /* even, return 1; odd, return 0 */
   return (n % 2) == 0 ? 1 : 0;
}
```